GitBook





Node, Express & libsass

a project from scratch workshop

Dale Sande

Table of Contents

Build a Node.js Project from Scratch

book passing

Node.js What's all the buzz about? Why are so many people talking about it? How can I get some of this awesome? Follow along in this workshop/tutorial to get your head wrapped around what it takes to make a Node project from scratch.

But it doesn't stop at Node. In the 'full stack JavaScript' world, there is a whole eco system of tools that you need to need to know about. Besides Node, there is Express, npm, Bower, Grunt, Gulp, etc .... This books's goal is not to deep dive into any one specific subject, but to provide the overview learning needed to build a good foundation.

Fun deck

I didn't make this deck, but it's an interesting and fun one to go through to talk about the interesting points of Node.js

Introduction to Node

There is more then enough documentation out there that supports the question, "Why Node?". Something that really rings true with me is not where Node is today, but where Node is going. Without a doubt, the Rails community has brought a lot to the table, but what made all those awesome ideas hard to swallow was the fact that they were locked inside Ruby. As amazing as Ruby is, not everyone wants to be a Ruby developer.

I particularly like this quote from *Why The Hell Would I Use Node.js? A Case-by-Case Introduction* [reference] by Tomislav Capan

> ... it's worth noting that Ryan Dahl, the creator of Node.js, was aiming to create real-time websites with push capability, "inspired by applications like Gmail". In Node.js, he gave developers a tool for working in the non-blocking, event-driven I/O paradigm.

Install Node

Before you run any installers, make sure you know what is on your computer. To see the version, simply run:

```
$ node --version
```

Of course to create and run a Node app, you need to have Node installed. To install Node, you can run the installer from their site.

Installing Node and npm is a great article on all the ways to get this set up. Pay attention to Step 4 where there are some pretty solid opinions on how to set things up.

A gist is made available that illustrates a series of ways to install node.

The article does state a personal opinion against using Homebrew. Brew has worked for me rather well, but this is an opinion you may need formulate on your own.

Node Package Manager (npm)

> npm is a NodeJS package manager. As its name would imply, you can use it to install node programs. Also, if you use it in development, it makes it easier to specify and link dependencies.

Learn more about npm

Depending on your install process you may or may not have NPM installed. To check, simply run:

```
$ npm --version
```

If you do not have npm installed, run the following:

Note: npm is the package manager for Node, so you can't use the package manager to install the package manager o_O

```
$ curl http://npmjs.org/install.sh | sh
```

Using npm

Now that you have npm installed, all registered packages are a simple command away. For a basic package install, run:

```
$ npm install <package>
```

This install method will install the package in a directory (`node_modules/`) relative to your project. At times you will need to install libraries globally so that you can access their code from any application that doesn't necessarily require them as a dependency.

To do this, you need to add the `-g` flag n the install process:

```
$ npm install -g <package>
```

Note: Depending on how Node is installed on your system, you may not have access to install a global package. To get around this, simply add the `sudo` command before the npm install method:

```
$ sudo npm install -g <package>
```

Using npm with a project

The most common use case for npm is to maintain a manifest of dependencies for your project. This is maintained with a package.json file.

You can either crate this file yourself, or there are ways to generate this file too. In any directory simply run `npm init` and the CLI will walk you through a series of questions and out put something like the following:

```
{
 "name": "toss-this",
 "version": "0.0.0",
 "description": "",
 "main": "index.js",
 "scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
 },
 "author": "",
 "license": "ISC"
}
```

Once you have this in your project adding to it using `npm install` is very easy. Simply add the `--save` flag to the command like the following:

```
$ npm install <package> --save
```

Adding Grunt to the project would update the `package.json` by adding a `dependencies` object in the json file:

```
{
 "name": "toss-this",
 "version": "0.0.0",
 "description": "",
 "main": "index.js",
 "scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
 },
 "author": "",
 "license": "ISC",
 "dependencies": {
  "grunt": "^0.4.5"
 }
}
```

Adding to this, if you wanted to add a dependency that is only for development versus production, you pass in the `-dev` flag:

```
$ npm install <package> --save-dev
```

Adding Gulp as a development dependency we get the following update to the `package.json` file by adding a `devDependencies` object:

```
{
  "name": "toss-this",
  "version": "0.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "grunt": "^0.4.5"
  },
  "devDependencies": {
    "gulp": "^3.6.2"
  }
}
```

Learn more about npm

npm is an amazingly complex utility when it comes to package management. See this npm cheatsheet for more in-depth information.

Learn more about package.json

The `package.json` has many features. To learn more about how this all works package.json An interactive guide is an amazing tool to learn from.

Maintaining dependencies

Unlike other package managers, npm installs these libraries directly into the root of your project. Without extra steps, these libraries will easily get committed to your version control.

For the most part you probably don't want to do this. Versioning is maintained via the `package.json` file and you shouldn't ever really go into the packages themselves and edit the code.

Using .gitignore

To keep npm libraries out of your version control, add the following to your .gitignore file.

```
node_modules
```

Getting the dependencies

The `package.json` file is maintaining your app's dependencies and you are not committing your dependencies to your Git repo, those who clone your project need to get these install these dependencies. Installing is very simple:

```
$ npm install
```

After executing that command you should see your CLI downloading the internet!

Express, the Node framework

Expressjs is the web application framework for Node.js apps.

> Express is a minimal and flexible node.js web application framework, providing a robust set of features for building single and multi-page, and hybrid web applications.

It has a wealth of APIs and is FAST AS HELL!!

Express is well known for NOT following after Rails as far as frameworks go, but more so it takes after another Ruby framework called Sinatra. The concept is simple, the framework gives you enough to get things up and running as fast as possible and all without getting in your way.

For the most part, Express continues to live up to this statement.

For this workshop, we will be using Express as the core tool for getting a web app up and running with a server, route support, error pages, loggers, etc ...

Install Express

Installing Express with npm is really easy. Keep in mind that there are two parts to Express, the library that runs it and an awesome generator.

To install Express:

```
$ npm install express -g
```

To install the generator:

```
$ npm install express-generator -g
```

Generator version

Express 4.0 was recently released and there are some who are not jiggy with that. So, npm has a way of specifying the version of generator you install.

```
$ npm install -g express-generator@3
```

Crate a new Express app

At this point, you should be able to go forth and create an app. In this example, we will create a Node.js app with Express framework.

```
$ express <your-new-app>
```

Running this command (using `demo-app` as the example), you should see the following:

```
 create : demo-app
 create : demo-app/package.json
 create : demo-app/app.js
 create : demo-app/public
 create : demo-app/public/javascripts
 create : demo-app/public/images
 create : demo-app/public/stylesheets
 create : demo-app/public/stylesheets/style.css
 create : demo-app/routes
 create : demo-app/routes/index.js
 create : demo-app/routes/users.js
 create : demo-app/views
 create : demo-app/views/index.jade
 create : demo-app/views/layout.jade
 create : demo-app/views/error.jade
 create : demo-app/bin
 create : demo-app/bin/www

 install dependencies:
   $ cd demo-app && npm install

 run the app:
   $ DEBUG=my-application ./bin/www
```

BOOM! Express takes care of most of the labor. Now, we do what the computer says, change directory into the app dir and run `npm install` .

What's in the app?

At this point, you should be able to do a `ls` and see the new structure that was created for you.

```
app.js    node_modules/ public/   views/
bin/    package.json  routes/
```

app.js

This is the logical starting point for your app. Some of the things in there, lets talk about:

The following lines, for this type of app, we don't need them:

```
var user = require('./routes/user');

app.get('/users', user.list);
```

Sets the path to the dir where the view files are located:

```
app.set('views', path.join(__dirname, 'views'));
```

Sets the path to the dir with static assets:

```
app.use(express.static(path.join(__dirname, 'public')));
```

Sets the root route for the app:

```
app.use('/', routes);
```

node_modules/

This is the dir where all your npm packages will reside.

public/

Directory for all static assets like images, JavaScript, CSS, fonts, etc ...

views/

Where all your layout and view Jade files will live.

bin/

There is a single file here, `www` and this is what activate the Node server.

package.json

Project description, scripts manager and the app manifest. Notice the following object:

```
"scripts": {
  "start": "node ./bin/www"
},
```

This is the code that allows you to run `npm start` for the app.

routes/

This is the directory where you will build out the RESTful routes for your app. With a base install there should be two files in there, `index.js` and `users.js`.

Fun with Routes

The `app.VERB()` methods provide the routing functionality in Express, where **VERB** is one of the HTTP verbs, such as `app.post()`. Multiple callbacks may be given, all are treated equally, and behave just like middleware, with the one exception that these callbacks may invoke `next('route')` to bypass the remaining route callback(s). This mechanism can be used to perform pre-conditions on a route then pass control to subsequent routes when there is no reason to proceed with the route matched.

The following snippet illustrates the most simple route definition possible. Express translates the path strings to regular expressions, used internally to match incoming requests. Query strings are **not** considered when peforming these matches, for example `GET /` would match the following route, as would `GET /?name=tobi`.

source

```
app.VERB(path, [callback...], callback)
```

Lets get into setting up some routes. In the `app.js` file the following line is how this comes together:

```
var routes = require('./routes/index');
```

What's happening here? Basically, Express is setting the `var` of `routes` to require the path and file of `./routes.index`.

This var is then used to set the root path of the app

```
app.use('/', routes);
```

Another thing we can do is `res.send()` and what ever we put in there will get sent directly to the browser. For example:

```
router.get('/foo', function(req, res){
  res.send('hello world');
});
```

Using the `res.send()` we can do fun things like even send in JSON objects.

```
router.get('/foo', function(req, res){
  res.send({'name':'Bob Goldcat', 'age': '41'})
});
```

This method allows us to then keep all our routes in the `index.js` file if needed. There are better ways to address a more complicated routing solution, but for the scope of this workshop, this is great.

What's in the index.js file?

Looking at our `index.js` file you should see the following:

```
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

router.get

This is the function that will 'get' the URL path of `/`. Then we need to create a function that will make a `req` (request) and `res` (response). The is another concept of `next` for chaining events would go here as well, but not shown in this example.

What is module.exports?

This is the object that's actually returned as the result of a require call. This is a Node feature, more on this at nodejs.org/api.

Build a new route

Looking at the syntax pattern, if we wanted to add a new route to the app, we can simply do something like the following:

```
router.get('/app', function(req, res) {
  res.render('app', { title: 'Express' });
});
```

Keep in mind that the value of the URL, `/app`, does not need to be the same value of the file itself. If the name of the view was `foo.jade` we could do the following:

```
router.get('/app', function(req, res) {
  res.render('foo', { title: 'Express' });
});
```

It's a route? It's a controller?

What's interesting about this is that the route function is containing logic. Inside the route is a `res.render` function:

```
res.render('foo', { title: 'Express' });
```

In the view template we see this:

```
h1= title
p Welcome to #{title}
```

These are two examples of how we can pull data from the 'controller/route' and display in the view. In this example we get the HTML output of:

```
<h1>Express</h1>
<p>Welcome to Express</p>
```

All of this seems to be a bleed of concerns as the route may also contain controller info? This is true and there is a

movement in the community to change the name of the dir from `routes` to `controllers`.

A great example of this can be seen at Holowaychuk's Express MVC example repo.

But for the sake of this workshop and consistency, we will keep the current convention.

404 errors?

Errors are already addressed in Express for you. In the `app.js` file, there is the following:

```
/// catch 404 and forwarding to error handler
app.use(function(req, res, next) {
    var err = new Error('Not Found');
    err.status = 404;
    next(err);
});
```

Then in the `views/` dir, there is `errors.jade`.

```
extends layout

block content
  h1= message
  h2= error.status
  pre #{error.stack}
```

Simple. If you want to customize your 404 page, simply edit this view.

Install and set up Grunt

Grunt is the workhorse of the Node world. You need tasks run, then you put Grunt to work. If you are from the Rails world, Grunt is much like make or Rake. There is a TOM of documentation out there on this task runner, but in this workshop, I will focus on the basics that will get you up and running with an application.

Grunt is pretty simple to set up, create a new `Gruntfile.js` file in the root of your project and add the following shell syntax:

```
module.exports = function(grunt) {
 grunt.initConfig({

   ...

 });

 grunt.loadNpmTasks('<package>');
};
```

To run Grunt, it is typical to install Grunt globally:

```
npm install grunt -g
```

BUT WAIT!! What about deployment? If we create tasks that require Grunt to run on the server, we need this to be a dependency on the app. So, let's install this as a local dependency:

```
npm install --save grunt
```

Using this base structure and Grunt installed, we can now begin to add new features to our app.

Need to learn more about Gulp

The following are some interesting articles that need to be worked into this workshop Gitbook.

Getting Started With Gulp

> This article will make the assumption that you have never used a task runner or command-line interface before and will walk through every step required to get up and running with gulp.

source

Roll Your Own Asset Pipeline with Gulp

> I've found myself using Gulp for just about everything involving HTML/CSS/JS these days. It's super fast, quick to write scripts for and flexible. I'm at a point now where I have a ton of projects I can just cd into, run gulp and be up and running. It's the best solution I've found for delivering static assets whether for local development or production.

source

Advanced Gulp File

> With gulp starting to find itself into my everyday workflow - I've started to understand its quirks and twists, and how to get along with it. My baseline gulfile.js has become a lot neater and advanced in its functionality that the one I originally developed back in March.

source

Add libsass

Sass, and it's port libsass, is the leading CSS pre-processor and by far the most feature rich. But one thing that separates libsass from others in the JavaScript community is that it's not written in JavaScript. It's actually written in C/C++. So, the same library is portable to just about any language that a wrapper can be written for. And by far the most popular wrapper is `node-sass`.

All of this doesn't really matter. The only thing this means is that we need to go through some additional, although extremely simple, setup steps.

First, let's install Node-Sass:

```
$ npm install --save node-sass
```

This will install the Node wrapper for Sass and the C/C++ libsass library. Next, in order for Grunt to make use of the library, we need to add the Grunt-Sass package.

```
$ npm install grunt-sass --save
```

To get all of this integrated into the project, we simply need to update the `Gruntfile.js`.

```
module.exports = function(grunt) {
  grunt.initConfig({
    sass: {
      dist: {
        files: {
          'public/stylesheets/style.css': 'sass/style.scss'
        }
      }
    }
  });

  grunt.loadNpmTasks('grunt-sass');
  grunt.registerTask('default', ['sass']);
};
```

In the `files:` object you will list the path to the output CSS and then the path to the input SCSS file.

Add the Sass

To get this running, we need to add the `sass` directory and put the `style.scss` file in there. In the root of the project:

```
$ mkdir sass
```

In the `sass` directory:

```
$ touch style.scss
```

And add the following Sass so that we know this is working:

```
$color: orange;

body {
 background-color: $color;
}
```

Run Grunt

At this point we are ready to run a `grunt` command and process some Sass.

Grunt-watch w/Livereload

Now that we have Grunt set up with Sass, when we run the `grunt` task in the CLI, this will process the edits we have. While in rapid development, going back the the CLI, typing in `grunt` and then refreshing the browser will get old fast.

Thankfully, we have options. `grunt-contrib-watch` is a npm package that will watch our Sass files and when one is touched, it will run the Grunt tasks to process it. To install:

```
npm install --save-dev grunt-contrib-watch
```

As an added bonus, not only can we run a watcher that will process our Sass, but we can also watch `.jade` files for changes as well.

In the `Gruntfile.js` we need to add some Grunt tasks:

```
module.exports = function(grunt) {
  grunt.initConfig({
    sass: {
      dist: {
        files: {
          'public/stylesheets/style.css': 'sass/style.scss'
        }
      }
    },
    watch: {
      source: {
        files: ['sass/**/*.scss', 'views/**/*.jade'],
        tasks: ['sass'],
        options: {
          livereload: true
        }
      }
    }
  });

  grunt.loadNpmTasks('grunt-contrib-watch');
  grunt.loadNpmTasks('grunt-sass');
  grunt.registerTask('default', ['sass']);
};
```

And there we have it. Grunt is installed, we have a functional `Gruntfile.js` and we can start writing some Sass too.

Add Livereload to layout.jade

In order to get Live reload to fire in the project we need to add a reference to a non-existent JavaScript file in the `layout.jade` file as the last thing in the `<body>` tag.

```
script(src='http://localhost:35729/livereload.js')
```

In a new Express project, it would look like this:

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content

    script(src='http://localhost:35729/livereload.js')
```

After this is added, go back and REFRESH the browser and then any edit in either Sass or Jade should fire the Livereload.

Bower all the things

Unless you live in a hole, you are well aware that the JavaScript revolution is all around us. Many of the amazing concepts I discovered in the Rails ecosystem are now bursting out into the JavaScript space allowing for a greater distribution of awesome. The three pillars are; Yeoman, Bower and Grunt.

The problem I need to solve is; what is the best way to get some library code up on Github and make it easily accessible to users without having to clone the project? Because, that's pretty lame, right?

Yeoman generators

Initially I came across generator-sass-boilerplate, a 'Yeoman generator for quickly scaffolding out Sass styles'. This is a very interesting approach for creating a complex library and allowing the user to customize the install. But for a simpler library of code, maybe just some functions and mixins, this is way to much overhead.

Bower is the answer

Fast forward to now. I recently came across new posts that really break down what Bower is and what it is best at. And it hit me, this IS the answer!

For those not in the know, Bower is an extremely simple solution for front-end package management.

> It offers a generic, unopinionated solution to the problem of front-end package management, while exposing the package dependency model via an API that can be consumed by a more opinionated build stack.

The beauty of Bower is held within it's simplicity. Bower has a registry, but it's not 100% necessary. The common command is `bower install <package>` where `<package>` can refer to a large number of options, thus making it dirt simple to just share some code. NICE!

Sticking with the 'dirt simple' theme, you can use Bower to easily pull a repo into your project without having to clone it. Even if it doesn't have a `bower.json` file.

For example Stipe, a Compass Extension library I wrote, is not Bower aware at all.

```
$ bower install git://github.com/Toadstool-Stipe/stipe.git
```

Run that command in any folder and you will pull in the entire repo with no Github history. This alone is pretty interesting.

Getting started with Bower

To get started, it's simple really. Assuming that Node and npm is installed on your local box, run:

```
$ npm install -g bower
```

Install Bower package

I won't go into exhaustive detail here, but 99% of the time you simply need to run:

```
$ bower install <package>
```

As stated above, there are alternate install options, but the primary solution is to have a `bower.json` file in the repo and

have it registered with Bower.

If you have a `bower.json` file in your project, explained in the next section, you can add the `--save` flag with the install and this will add the library as a dependency in your project. Sweet.

When you distribute the project, a user who clones it only has to run `$ bower install` and it will pull in all the external resources. Nice!

Commit or not to commit!?

This new system of creating and distributing resources raises an interesting question; do you commit all your bower packages or not? In the Ruby world, Gems are not actually part of the project, but dependencies of the project, and never committed to the project's version control. In this new JavaScript world, Node and Bower package dependencies are referenced via a manifest, much like the Gemfile in Ruby, but are actually installed into the root of the project directory.

There is a whole discussion on this topic. I look at it this way, when you install a Bower library, are you leaving this as a dependency or are you making modifications?

The choice is up to you, the arguments are strong on either side. In a situation where you are actually forking the code you installed, then the answer is pretty clear, it should be committed to the project or you need to fork the dependency.

Generate new Bower package

Creating a new Bower package is again, really simple.

```
$ bower init
```

In the CLI, this will initiate a series of questions, the answers of which will be plugged into the new `bower.json` file it creates. Put as much in as you want, but all you really need is:

```
{
  "name": "your-project",
  "version": "0.1.0"
}
```

And that's it really. You have just created your first Bower resource library. Now go forth and build! Build out your resources, documentation, etc ... your package is ready to go at any time.

For easy testing, remember the `$ bower install git://github.com/ ...` trick? Run this against a new repo and see how it installs.

Be mindful of this step and what the package contains. In my opinion, I see Bower as a great way to distribute smaller, specific libraries of code. When I pull in your Bower package, do I really want all your documentation, tests, demonstration resources, etc ... As an example, I am going to pick on the Bourbon kids here, run:

```
$ bower install bourbon
```

Running this installer, you will get the whole repo. I don't want the whole repo, all I really want is what is in the `dist/` dir. To solve this, another developer forked Bourbon and created a new repo called bower-bourbon:

```
$ bower install bower-bourbon
```

Running this install you actually only get what is in the `dist/` dir. But are these forks reliable? Ohh open source, you are a

wild one.

UPDATE: It's been brought to my attention that using the Bower install of Bourbon pulls in it's 3.2 Beta and appears not to be fully functional. This section was not intended to say "bad Bourbon" but to simply illustrate that in some cases, using Bower, you will get more of the library then you really want.

Bower registration

Once you are ready for release, register it with Bower. The criteria is pretty simple:

1. Make sure your repo has the `bower.json` file
2. You must use semantic versioning
3. Your package must be available at a Git endpoint, e.g. Github

Once you have all of that, run this command with your new package name and the Git endpoint:

```
$ bower register <my-package-name> <git-endpoint>
```

Registration is painless. Once you get the green light on everything, give it a test and do a `$ bower install <my-package-name>`

Bower and Sass

Bower and Sass libraries are an amazing pairing. There are small repos all over Github where the complexity of making them a Ruby Gem/Compass Extension was just to much overhead. You are required to either fork, clone, or god forbid, copy and paste code into your project. What? Are we not civilized?

In the Ruby world, developers are used to having Gems and Compass Extensions installed in a safe, *untouchable location. The new Gem is added to the Gemfile and we simply reference the library in the project.

***Untouchable is a frustration with many developers. Importing Sass libraries that they did not have control over, or were unable to modify, that actually output CSS can be very frustrating.**

In the new JavaScipt world, the library is added to the `bower.json` manifest or simply installed, but instead of it being in a obscured location, it is installed into the root of the project. This keeps things simple from an install perspective, but this means our Sass imports are in relative directories. Not a big deal, but different from what we are used to.

So, what does a Sass Bower package look like? Let's take a simple project I created called, sass-icon-fonts. This package is simply a couple of mixins, one that allows the developer to easily create a `@font-face` set of rules and another is the ability to quickly generate a series of icon-font rules. The mini library comes with instructions and a very simple API.

Now, let's imagine you are building a Node project and you want to use this package as a resource, run:

```
$ bower install sass-icon-fonts --save
```

This installs the package and adds the dependency to your `bower.json` file. Located at the root level of the project is your `sass/` directory, within that is your `application.scss` file. At your root is the `bower_components` directory. For your `application.scss` file to access the new library, you will need to import a relative path to the library, as illustrated in the following:

```
@import "../bower_components/sass-icon-fonts/_ico-font.scss";
```

While the previous example works, while I found this acceptable, I didn't really find it awesome. Digging more into the

Grint-Sass API I discovered the includePaths function. This allows you to set an import path to include.

```
options: {
  includePaths: [
    './bower_components/bower-bourbon'
  ]
}
```

Now that you have this in your Grunt file, you can simply reference the library's main manifest file with a simply Sass import, like so:

```
@import "bourbon";
```

Bower in your npm

One of the things that I find slightly annoying about using Bower is that I have to run separate commands when initialing a new project. I already have to use npm, can't I just bind these things together?

Yes, yes you can. In your `package.json` file, simply extend the `scripts` object and pass in the the `bower install` command. This is why I really LOVE this stuff!

```
"scripts": {
  "install": "bower install"
}
```

Now, when you run `npm install` this will not only install all your Node packages, but install your Bower packages as well. NICE!

Bower behind the firewall

If you find yourself behind a firewall that does not allow for the `git://[repo]` protocol, there is a fix for this. First, I suggest maually doing a clone using the `https://[repo]` protocol to make sure that this is really the issue. If the `https://[repo]` protocol works, then you may want to make the following update:

```
git config --global url."https://"
```

Thank you Stack Overflow!

Summary

When I say I want to Bower all the things, I mean just that. Now understanding Bower, I am looking at simple package management in a whole new light and I hope that you do to.

No more forking, cloning, deleting `.git/` directories just to include a library into a project. I am looking at creating Sass modules in a whole new light as well. Not that Compass extensions were difficult, but Bower frees me of multiple dependencies. Something that has been a real issue on many projects.

Bower - Grunt - Sass

Now that we know the powers of Bower to easily manage our front-end development dependencies, what do we need to do to add a Bower package of Sass code to our project?

Bower install

First off, let's install a simple Bower package for illustration:

```
$ bower install css-calc-mixin --save
```

There, we now have the library of code in our project.

Update Gruntfile.js

Next we want to update the `Gruntfile.js` so that we can easily include the library into our Sass files. Without this step, we would need to write fill paths in our Sass file to this and that's simply lame.

In the Grunt-Sass API we have options and the one we need to use is `includePaths`. Here we can pass in a string that is the full path from root to the Bower package into an array.

```
module.exports = function(grunt) {
  grunt.initConfig({
    sass: {
      dist: {
        files: {
          'public/stylesheets/style.css': 'sass/style.scss'
        }
      },
      options: {
        includePaths: [
          './bower_components/css-calc-mixin'
        ]
      }
    },
    watch: {
      source: {
        files: ['sass/**/*.scss', 'views/**/*.jade'],
        tasks: ['sass'],
        options: {
          livereload: true, // needed to run LiveReload
        }
      }
    }
  });

  grunt.loadNpmTasks('grunt-contrib-watch');
  grunt.loadNpmTasks('grunt-sass');
  grunt.registerTask('default', ['sass']);
};
```

Update style.scss

To make use of this new Bower package library, we simply need to use a Sass convention to import the code.

```
@import "css-calc-mixin";
```

To test that this is working, let's add a little bit of code that references the Bower library.

```
.block {
  @include calc(width, 220px);
}

.block {
  @include calc(margin, 220px, 0);
}

.block {
  @include calc(width, 220px, true, 0);
}

.block {
  @include calc(width, 220px, true, 0, 50%, '+');
}
```

Back in the CLI, run `grunt` and we should see green lights all day long!

Add some data

Great demo that adds the next layer of awesome to this workshop. Adding a Mongo DB to the project to make a simple i/o UX.

> My favorite thing about MongoDB is that it uses JSON for its structure, which means it was instantly familiar for me. If you're not familiar with JSON, you'll need to do some reading, as I'm afraid that's outside the scope of this tutorial.
>
> Let's add a record to our collection. For the purposes of this tutorial, we're just going to have a simple database of usernames and email addresses. Our data format will thus look like this:

```
{
 "_id" : 1234,
 "username" : "cwbuecheler",
 "email" : "cwbuecheler@nospam.com"
}
```

Anyone out there willing to help put the rest of this together, I would love a Pull Request!

Heroku

The following are basic steps needed to quickly set up a Node.js app with Express and deploy to Heroku.

Step 1 - Heroku account

Make sure you have a Heroku account set up

Step 2 - Install Heroku Toolbelt

Download and install the tool belt package specific for your OS

OSX

https://toolbelt.heroku.com/osx

Windows

https://toolbelt.heroku.com/windows

Linux

```
$ wget -qO- https://toolbelt.heroku.com/install-ubuntu.sh | sh
```

Step 3 - Log into you account

Once toolbelt is installed, you should be able to access your account

```
$ heroku login
Enter your Heroku credentials.
Email: adam@example.com
Password:
Could not find an existing public key.
Would you like to generate one? [Yn]
Generating new SSH public key.
Uploading ssh public key /Users/adam/.ssh/id_rsa.pub
```

That's pretty much about it. Pass these steps and you are ready to DEPLOY THE CODES!

Deploy your first app

At this point, you should have an app that works locally on your computer. The following steps outline updates you need to make in order to deploy the codes.

Update package.json

In this step, we need to add some code to the `package.json` file so that we can run the app from a remote server.

Right now, there is a good chance that the file will look like this:

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "express": "~4.0.0",
    "static-favicon": "~1.0.0",
    "morgan": "~1.0.0",
    "cookie-parser": "~1.0.1",
    "body-parser": "~1.0.0",
    "debug": "~0.7.4",
    "jade": "~1.3.0"
  }
}
```

At the end of the `dependencies": { ... }` object, you need to add a comma `,` so that we can add more code. First let's add the `main` keyword:

```
"main": "app.js",
```

Notice the trailing comma? This is because we are going to add more stuff. After that, add in the `engines` object and the specific engines we need to run this app:

```
"engines": {
  "node": "0.10.26",
  "npm": "1.4.3"
}
```

You should have something that looks like the following:

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "express": "~4.0.0",
    "static-favicon": "~1.0.0",
    "morgan": "~1.0.0",
    "cookie-parser": "~1.0.1",
    "body-parser": "~1.0.0",
    "debug": "~0.7.4",
    "jade": "~1.3.0"
  },
  "main": "app.js",
  "engines": {
    "node": "0.10.26",
    "npm": "1.4.3"
  }
}
```

Don't forget the Grunt + Bower

If at this time you do not have any of the Grunt packages or Bower in the `dependenciess` object, we need to get that in there.

You can either add them manually to the `package.json` file or dun:

```
$ npm install --save grunt
$ npm install --save grunt-sass
$ npm install --save bower
```

Something that you probably don't have is the ability for the deployed server to run the Grunt tasks. For this we need Grunt-CLI.

```
$ npm install --save grunt-cli
```

Right about now, you should be looking pretty good.

Postinstall instructions

When we deploy the codes to Heroku, we have to tell it to run some commands, basically install the Bower packages and run the Grunt tasks. To do this, we need to add the instructions within the `scripts` object of the `package.json` file.

Directly under the `"start": "node ./bin/www"`, add:

```
"postinstall": "./node_modules/.bin/bower install && ./node_modules/.bin/grunt"
```

There, now we have everything that Heroku needs to install the packages and run the scripts.

Add the Procfile

This is a file that Heroku needs in order to start the app.

```
$ touch Procfile
```

Add the following code:

```
web: npm start
```

Heroku will use this to kick start the app.

Make this a Git repo

It is important to make this a git repo BEFORE you create the Heroku server. **WAIT!** Before you go all crazy on the Git, there are some things we need to do.

You should have a `.gitignore` file in your repo at this point. open that up and make sure you are ignoring all the `node_modules`, all the `bower_components` and anything in the `/stylesheets/*.css` spectrum.

```
node_modules
public/stylesheets/*.css
bower_components
```

Great. Now you can `git init` your repo.

```
$ git add .
$ git commit -m "initial commit"
```

It is not required at this time to make this a Github repo, but you may want to do this if you make this a real app.

Deploy the codes

This is pretty hard here. Make sure to follow the commands specifically:

```
$ heroku create <your-app-name>
$ git push heroku master
```

Rejoice

If all is well, you should see a return like this:

```
http://<your-new-app>.herokuapp.com/ deployed to Heroku
```

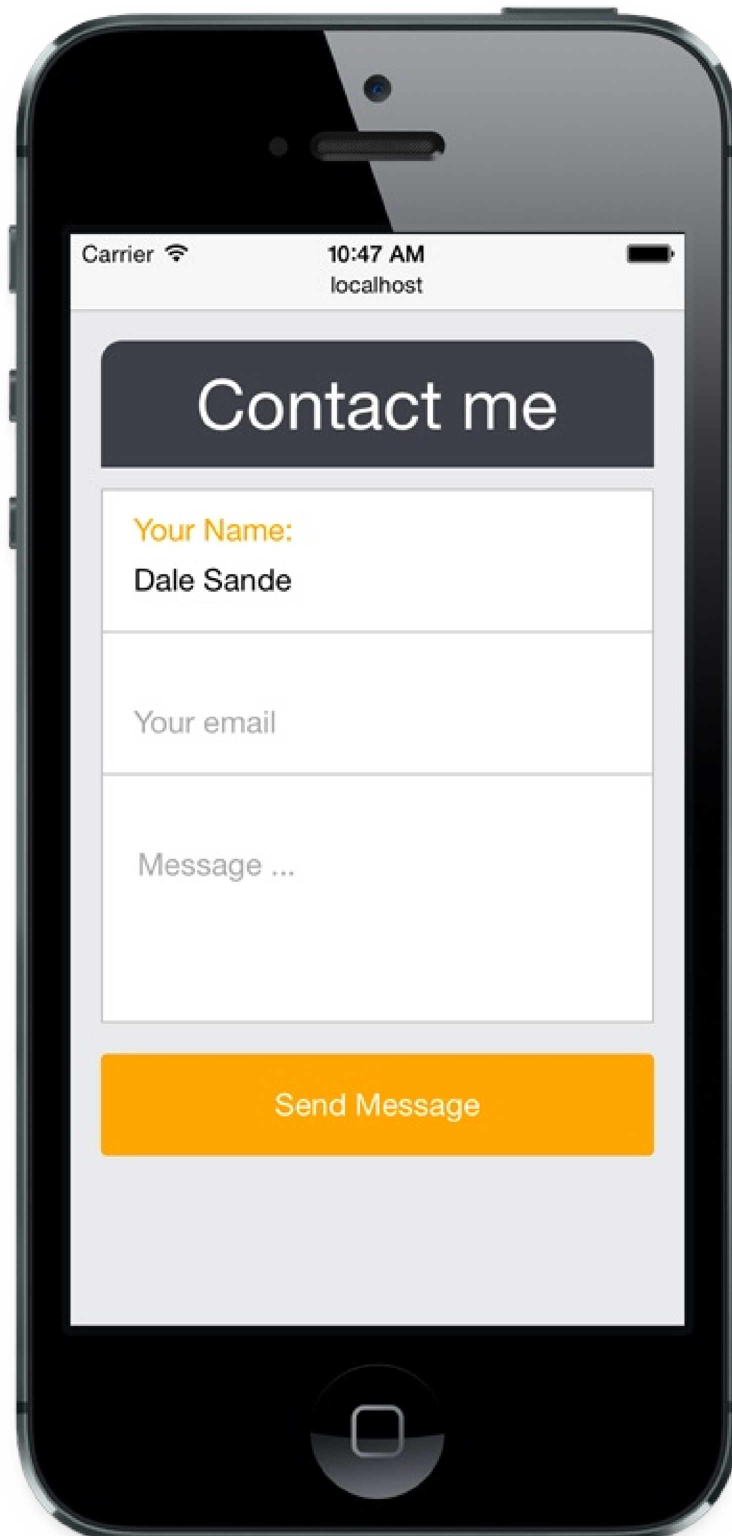
Go to that URL and REJOICE!!!!!

Build a message form:

step-by-step directions

From 0 - 60, build a contact widget From 0 - 60, build a contact widget.

The following demo is a re-do of a popular UI/UX pattern of revealing the form label as the user enters data into the form.



Getting libsass set up

Install libsass with Grunt Sass and add Bourbon. Node-Sass is installed as a dependency of Grunt-Sass

```
$ npm install grunt-sass --save
$ npm install node-bourbon --save
```

Add bower.json file

```
{
  "name": "class-demo",
}
```

Add some Bower packages

```
$ bower install color-scale --save
$ bower install type-rhythm-scale --save
$ bower install rwd-toolkit --save
```

Install Grunt

```
npm install grunt --save
```

Install Grunt Watch

```
npm install grunt-contrib-watch --save-dev
```

Add `gruntfile.js`

```
module.exports = function(grunt) {
  grunt.initConfig({
    sass: {
      dist: {
        files: {
          'public/stylesheets/application.css': 'sass/application.scss'
        },
        options: {
          sourceMap: true,
          includePaths: [
            require('node-bourbon').includePaths,
            './bower_components/color-scale',
            './bower_components/type-rhythm-scale',
            './bower_components/rwd-toolkit'
          ]
        }
      }
    },
    watch: {
      source: {
        files: ['sass/**/*.scss', 'views/**/*.jade'],
        tasks: ['sass'],
        options: {
          livereload: true, // needed to run LiveReload
        }
      }
    }
  });

  grunt.loadNpmTasks('grunt-contrib-watch');
  grunt.loadNpmTasks('grunt-sass');
  grunt.registerTask('default', ['sass']);
};
```

Create new Sass file

Create the following directory and file, then add some sass to the file.

```
$ mkdir sass
$ touch sass/application.scss
```

Install library dependencies

Add these to the `application.scss` manifest to make Sass aware of these dependencies.

```
@import "bourbon";
@import "type-rhythm-scale";
@import "rwd-toolkit";
```

Get grunt started

```
$ grunt
$ grunt watch
```

Version control

Add `.gitignore` file and add the following:

```
# OS generated files
####################
.DS_Store
.DS_Store?
._*
.Spotlight-V100
.Trashes
ehthumbs.db
Thumbs.db

# Generated CSS output
#######################
public/stylesheets/*.css
public/stylesheets/*.css.map

# Package dependencies
######################
node_modules/
bower_components
```

Add in version control

```
$ git add --all
$ git commit -m "add all the things"
```

Update the layout

Update `layout.jade` to use `application.css`

```
link(rel='stylesheet', href='/stylesheets/application.css')
```

Add in Live Reload

```
script(src="//localhost:35729/livereload.js")
```

Refresh the browser and then begin making small edited in the Sass and Jade files to make sure that Live Reload is working.

Build the view

Starting with the layout, get more stuff in there to make this work correctly

```
meta(charset='utf-8')
meta(http-equiv='X-UA-Compatible', content='IE=edge')
meta(name='description', content='#{description}')
meta(name='viewport', content='width=device-width, initial-scale=1.0, minimum-scale=0.5 maximum-scale=1.0 minimal-ui')
```

Update the `index.js` file in `./routes`

```
res.render('index', { title: 'Contact me', description: 'This is a new demo' });
```

Open `./views/index.jade` and add the following:

```
section.message-container
  h1.title= title
  form#form.form(action='#', method='get')
    ul
      li
        label(for='name') Your Name:
        input#name(type='text', placeholder='Your Name', name='name', tabindex='1')
      li
        label(for='email') Your Email:
        input#email(type='email', placeholder='Your Email', name='email', tabindex='2')
      li
        label(for='message') Message:
        textarea#message(placeholder='Message…', name='message', tabindex='3')

    button#submit Send Message
```

Build the UI config file

Create the file

```
$ touch _config.scss
```

Add the following code:

```
/////// Typography configuration
// *----------------------------------------
$font-size: 16;

$heading-1: 36;
$heading-2: 32;
$heading-3: 28;
$heading-4: 18;
$heading-5: 18;
$heading-6: 18;

$line: $font-size * 1.5;
$small-point-size: 10;
$large-point-size: 14;

$primary-font-family: #{"Helvetica Neue", Arial, sans-serif};
$secondary-font-family: #{"Helvetica Neue", Arial, sans-serif};
$heading-font-family: #{"Helvetica Neue", Arial, sans-serif};


/////// Default webfont directory
// *----------------------------------------
$fontDir: "fonts/";

/////// Default image directory
// *----------------------------------------
$imgDir: "images/";

/////// OOCSS generic base colors
// *----------------------------------------
$alpha-primary:   #5a2e2e;        // red
$bravo-primary:   #3e4147;        // green
$charlie-primary: #fffedf;        // yellow
$delta-primary:   #2a2c31;        // blue
$echo-primary:    #dfba69;        // accent

$alpha-gray:      #333;           //black

/////// Color math
// *----------------------------------------
@import "color-scale";


/////// Semantic variables
// *----------------------------------------
// abstract 'white' value to easily applied to semantic class objects
$white:                               #fff;

// primary header font color
$primary-header-color:                $alpha-gray;

// default heading font weight
$heading-font-weight:                 normal;

// primary font color for the app
$primary-text:                        $alpha-gray;

// default `href` link color
$href-color:                          $delta-color;

// default shadow colors
$shadow-color:                        fade-out($alpha-color, 0.5);

// default border color
$border-color:                        $alpha-color;
```

```
/////// HTML 5 feature colors
// *----------------------------------------
// used with the `ins` tag
// http://www.w3schools.com/tags/tag_ins.asp
$ins-color:                             $charlie-color;

// used with the `mark` tag
// http://www.w3schools.com/html5/tag-mark.asp
$mark-color:                            $charlie-color;

// webkit tap highlight color
$webkit-tap-hightlight:                 $delta-color;

// overrides the default content selection color in the browser
$selection-color:                       $charlie-color;
$selection-text-color:                  $primary-text;



//////// Default animation properties
// *----------------------------------------
$trans: .333s ease;
```

Add to the `application.scss` file

```
/////// App Config - this is where most of your magic will happen
// *----------------------------------------
@import "config";
```

Add Reset

Add the file

```
$ touch _reset.scss
```

Add to the `application.scss`

```
/////// Standard CSS reset stuff here
// *-------------------------------------------------
@import "reset";
```

Add the following code:

```
// * Let's default this puppy out
// *-------------------------------------------------

html, body, div, span, object, iframe, h1, h2, h3, h4, h5, h6, p, blockquote, pre, abbr, address, cite, code, del, dfn, em, img, ins, kbd, c
  margin: 0;
  padding: 0;
  border: 0;
  vertical-align: baseline;
  background: transparent;
}

* {
  -moz-box-sizing: border-box;
  box-sizing: border-box;
}

body {
  font-size: 100%;
  -webkit-font-smoothing: antialiased;
}

article, aside, figure, footer, header, hgroup, nav, section {
  display: block;
}

// * Responsive images and other embedded objects
// * Note: keeping IMG here will cause problems if you're using foreground images as sprites, like, say for Google Maps custom placen
// * There has been a report of problems with standard Google maps as well, but we haven't been able to duplicate or diagnose the is

img, object, embed {
  max-width: 100%;
}

img {
  border-style: none;
  border-color: transparent;
  border-width: 0;
}

// * we use a lot of ULs that aren't bulleted.
// * don't forget to restore the bullets within content.

ol,ul {
  list-style: none;
}

blockquote, q {
  quotes: none;

  &:before, &:after {
    content: '';
    content: none;
  }
}

a {
  margin: 0;
  padding: 0;
```

```
  font-size: 100%;
  vertical-align: baseline;
  background: transparent;
  &:focus {
    text-decoration: underline ;
    outline: none;
  }
}

del {
  text-decoration: line-through;
}


pre {
  //white-space: pre
  // * CSS2
  white-space: pre-wrap;
  // * CSS 2.1
  //white-space: pre-line
  // * CSS 3 (and 2.1 as well, actually)
  word-wrap: break-word;
  // * IE
}

input {
  &[type="radio"] {
    vertical-align: text-bottom;
  }
}

input, textarea, select, button {
  font-family: inherit;
  font-weight: inherit;
  background-color: #fff;
  border: 0;
  padding: 0;
  margin: 0;
}

table {
  font-size: inherit;
}

sub, sup {
  font-size: 75%;
  line-height: 0;
  position: relative;
}

sup {
  top: -0.5em;
}

sub {
  bottom: -0.25em;
}

// * standardize any monospaced elements

pre, code, kbd, samp {
  font-family: monospace, sans-serif;
}

input {
  &[type=button], &[type=submit] {
    @extend %stipe-cursor-pointer;
  }
}

button {
  cursor: pointer;
  margin: 0;
  width: auto;
  overflow: visible;
}

a.button {
  display: inline-block;
}

// * scale images in IE7 more attractively
```

```
.ie7 img {
  -ms-interpolation-mode: bicubic;
}

// * Ok, this is where the fun starts.
// *-----------------------------------------------

a:link {
  -webkit-tap-highlight-color: $webkit-tap-hightlight;
}

ins {
  background-color: $ins-color;
  color: black;
  text-decoration: none;
}

mark {
  background-color: $mark-color;
  color: $primary-text;
  font-style: italic;
  font-weight: bold;
}

::selection {
  background: $selection-color;
  color: $selection-text-color;
}

::-moz-selection {
  background: $selection-color;
  color: $selection-text-color;
}
```



Start with the global layout

```
$ mkdir layouts
$ touch layouts/_global.scss
$ touch layouts/_manifest
```

Add to _global.scss

```
body {
  background-color: $delta-scale-juliet;
}
```

Add to application.scss

```
/////// Layouts
@import "layouts/manifest";
```

Create a module

In the Sass directory, lets create a module directory with the necessary files inside:

```
$ mkdir modules
$ mkdir modules/message-container
$ touch modules/message-container/_module-message-container.scss
$ touch modules/message-container/_manifest.scss
```

Add the following to the `_module-message-container.scss` file:

```
.message-container {
  margin: 1em auto;
  width: 90%;
  padding-bottom: 100px;
  @media #{$tablet} {
    width: 75%;
  }
  @media #{$desktop} {
    width: 50%;
  }
}
```

Add to _manifest.scss

```
@import "module-message-container";
```

Add to application.scss

```
/////// Modules
@import "modules/message-container/manifest";
```

Central module manifest

In the `application.scss` we could enter each module on-by-one as described above, but we could also add a manifest at the root of `modules` that will import all the manifests contained within.

So in the `application.scss` we do the following:

```
/////// Modules
@import "modules/manifest";
```

Then in `modules/manifest.scss` we do this:

```
/////// Sub-Modules
@import "message-container/manifest";
```

This helps keep things easier to manage as the we never need to update the `application.scss` file and at the root of the modules directory, where we are working, we just need to add a new listing there. Everything is imported and everything just works.

Our folder structure would look like the following:

```
|- application.scss
|--- modules/
|----- _manifest.scss
|----- message-container/
|------- _manifest.scss
|------- _module-message-container.scss
```

Typography

Create new sass file

```
$ touch _typography.scss
```

Add the following code:

```
html {
  font: em($font-size, 16) $primary-font-family;
  line-height: baseline($font-size);
  color: $primary-text
}

h1, h2, h3, h4, h5, h6, [role=heading] {
  @include heading();
}

.title {
  margin-bottom: em(5);
  padding: 0.25em 0;
  text-align: center;
  background-color: $delta-scale-bravo;
  color: $white;
  border-radius: em(5) em(5) 0 0;
}
```

Add below the reset in application.scss

```
/////// Base
@import "typography";
```

Forms

Create a new sass file

```
$ touch _forms.scss
```

Create a new forms directory

```
$ mkdir forms
$ midir forms/extends
```

Add the following files:

```
$ touch forms/_manifest.scss
$ touch forms/extends/_default-inputs.scss
$ touch forms/extends/_display-block.scss
$ touch forms/extends/_manifest.scss
```

Add the following to `forms/extends/_manifest.scss`:

```
@import "default-inputs";
@import "display-block";
```

Add the following to `forms/extends/_default-inputs.scss`:

```
%default-inputs {
  width: 100%;
  height: 100%;
  padding: 2.25em 1em 1em;
  outline: none;
  font-size: 1em;
}
```

Add the following to `forms/extends/_display-block.scss`:

```
%display-block {
  width: 100%;
  display: block;
}
```

Add the following to `forms/_manifest.scss`:

```
@import "extends/manifest";
```

Add the following to `_forms.scss`:

```
@import "forms/manifest";

.form {
  ul {
    border-bottom: 1px solid $hotel-gray;
    background-color: $white;
    margin-bottom: 1em;
  }
  li {
    border: 1px solid $hotel-gray;
    border-bottom: 0;
    position: relative;
  }
}

label {
  @extend %display-block;
  position: absolute;
  font-size: em(16);
  top: .5em;
  left: 1em;
  color: orange;
  opacity: 1;
  transition: #{$trans} top, #{$trans} opacity;
  .js-hide-label & {
    opacity: 0;
    top: 1.5em;
  }
}

input[type="text"] {
  @extend %display-block;
  @extend %default-inputs;
}

input[type="email"] {
  @extend %display-block;
  @extend %default-inputs;
}

textarea {
  @extend %display-block;
  @extend %default-inputs;
  height: 8em;
  resize: none;
}
```

Buttons

Add the following file:

```
$ touch _buttons.scss
```

Add to `application.scss` under the forms import

```
@import "buttons";
```

Open and add the following code:

```
button {
  @extend %default-inputs;
  -webkit-appearance: none;
  background: orange;
  color: white;
  border-radius: em(3);
  padding: 1em;
}
```

Add some jQuery magic

Open `layout.jade` and add in the foot of the doc:

```
script(src='//ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js')
script(src="/javascripts/app.js")
```

Create the following file:

```
$ touch public/javascripts/app.js
```

Open `app.js` and add the following:

```
// Test for placeholder support
$.support.placeholder = (function(){
  var i = document.createElement('input');
  return 'placeholder' in i;
})();

// Hide labels by default if placeholders are supported
if($.support.placeholder) {
  $('.form li').each(function(){
    $(this).addClass('js-hide-label');
  });
}

$('.form li').find('input, textarea').on('keyup blur focus', function(){

  // Cache our selectors
  var el = $(this),
    parent = el.parent();

    // Add or remove classes
    if( el.val() === '' ) {
        parent.addClass('js-hide-label');
    } else {
        parent.removeClass('js-hide-label');
    }
});
```